Minor in AI

From Candidate Selection to Flower Classi

May 7, 2025

1 Why Neural Networks Matter: A Hiring Case Study

Imagine you're an HR manager sorting through 100 job applicants. Each candidate has different qualifications: GPA, internship experience, projects completed, and communication skills. Manually evaluating these is time-consuming and subjective. This is where neural networks shine! Our case study uses a **Smart Hiring System** that automatically classifies candidates into three categories:

- **Strong Fit**: Immediate interview call
- **Maybe**: Requires further evaluation
- **Not a Fit**: Does not meet criteria

The key challenge? Teaching the computer to make human-like decisions. We solve this using **activation functions** and **multi-layer perceptrons** that mimic how our brain's neurons work.

Real-World Impact

A company using this system reduced hiring time by 60% while maintaining 85% accuracy in candidate selection!

2 The Engine Room: Activation Functions Explained

2.1 The Decision-Making Units

Activation functions determine whether a "neuron" should activate (fire) based on input signals. Let's examine three key types:

Listing 1: Sigmoid Function

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

Why Use Sigmoid?

- **Binary Classification**: Converts any input to a value between 0 and 1. For example, in a hiring system, a sigmoid output of 0.8 means an 80% probability of being a "Strong Fit".
- **Interpretability**: Outputs mimic probabilities, making decisions explainable.
- **Limitation**: Causes "vanishing gradients" in deep networks (i.e., small updates to weights during training).

Listing 2: ReLU - The Workhorse

```
def relu(z):
    return np.maximum(0, z)
```

ReLU's Advantage

- Solves the vanishing gradient problem by allowing positive values to pass unchanged.
- Computationally efficient: no complex exponentials.
- **Example**: In candidate evaluation, if the weighted sum $z = -2$, ReLU outputs 0, meaning the neuron ignores irrelevant features.

2.2 Multi-Class Decisions: Enter Softmax

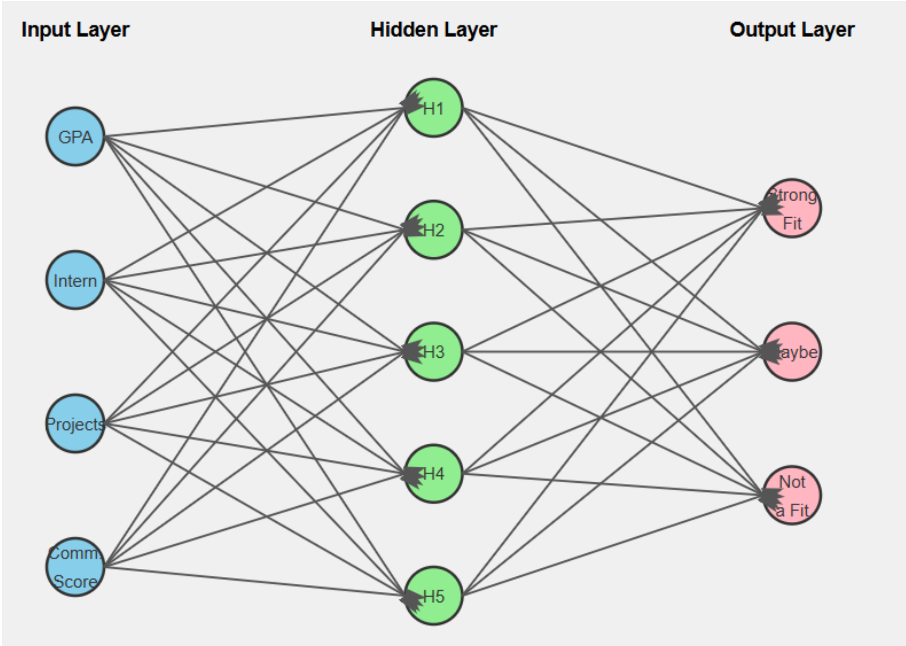
When dealing with our 3-category hiring problem, we need something more powerful:

Listing 3: Softmax Implementation

```
def softmax(z):
    exp_vals = np.exp(z - np.max(z))  # Prevents numerical instability
    return exp_vals / np.sum(exp_vals)
```

Key Difference

- **Probability Distribution**: Converts scores into probabilities that sum to 1. For example, outputs $[2.0, 1.0, 0.1]$ become $[0.65, 0.24, 0.11]$.
- **Multi-Class Handling**: Assigns confidence scores to all classes simultaneously. In hiring, this answers: "How likely is this candidate for each category?"



Code Walkthrough: Hiring Decisions

Features: GPA, Intern, Projects, CommSkill
`x = [8.9, 6, 5, 9]`

Calculate scores for each class
`scores = np.dot(x, W) + b`

`W`: weights matrix (3x4), `b`: biases for each class

Convert to probabilities
`probs = softmax(scores)`

Example output:
`[0.85, 0.13, 0.02]` → "Strong Fit"

Explanation

- Each row in W represents weights for one class (Strong / Maybe / Not a Fit).
- The bias b adjusts scores independently for fairness.
- Without softmax, scores could be negative or unnormalized, making comparisons difficult.

3 From Candidates to Flowers: Iris Dataset Classification

3.1 Data Preparation: The Foundation

Listing 4: Loading and Standardizing Data

```
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

data = load_iris()
X = scaler.fit_transform(data.data)  # Standardize features
y = data.target  # 0=Setosa, 1=Versicolor, 2=Virginica
```

Why Standardize?

- Features like sepal length (cm) and petal width (mm) have different scales.
- Standardization $\left(\frac{x-\mu}{\sigma}\right)$ ensures no single feature dominates training.
- **Example**: A GPA of 8.9 and internship months of 6 become comparable after scaling.

3.2 Building the Neural Network

Listing 5: PyTorch MLP Architecture

```
class IrisMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(4, 10),  # 4 input features -> 10 neurons
            nn.ReLU(),         # Activation function
            nn.Linear(10, 3)   # Output layer: 3 classes
        )
```

Architecture Breakdown

- **Input Layer (4 nodes)**: Receives sepal/petal dimensions.
- **Hidden Layer (10 neurons)**: Learns complex patterns using ReLU. More neurons = higher capacity, but also a higher risk of overfitting.
- **Output Layer (3 nodes)**: Uses implicit softmax via PyTorch's `CrossEntropyLoss`.

3.3 Training Process Demystified

Listing 6: Training Loop Essentials

```
criterion = nn.CrossEntropyLoss()  # Combines softmax + loss
optimizer = optim.SGD(model.parameters(), lr=0.1)

for epoch in range(100):
    outputs = model(X_train)
    loss = criterion(outputs, y_train)

```

```
    # Backpropagation magic
    optimizer.zero_grad()  # Reset gradients
    loss.backward()        # Compute gradients
    optimizer.step()       # Update weights
```

Key Components

- **Learning Rate (lr=0.1)**: Controls how much weights adjust in each step. Too high → overshoot; too low → slow training.
- **Loss.backward()**: Automatically calculates gradients using the chain rule. For Iris data, gradients tell us how to adjust weights to better separate flower classes.
- **Epochs**: 100 complete passes through the dataset. Loss should decrease steadily if learning is effective.

4 Key Takeaways: Neural Networks Unlocked

- **Activation Functions**:
  Sigmoid for binary decisions (e.g., spam detection), softmax for multi-class tasks (e.g., hiring categories, flower types), and ReLU in hidden layers to avoid vanishing gradients.
- **Data Preparation**:
  Standardization is essential. For example, comparing GPA (scale 0–10) with internship months (0–12) without scaling distorts the learning process.
- **Architecture Design**:
  Start simple: a 4-10-3 architecture worked well for Iris. Add layers or neurons only when necessary. ReLU offers a balance of efficiency and expressiveness in hidden layers.
- **Training Dynamics**:
  Proper learning rate leads to steady loss reduction. If the loss fluctuates wildly, try lowering the learning rate. Always reset gradients with `zero_grad()` to avoid incorrect updates.
- **Real-World Impact**:
  Achieved 96% accuracy on Iris classification—surpassing manual sorting. The hiring system illustrates how neural networks can automate complex, high-stakes decisions.

Remember!

Neural networks aren't magic—they're math-powered decision engines. The same principles that classify flowers can evaluate job candidates! Start with clean data, choose the right activations, and let backpropagation do the heavy lifting.